共达地 gddeploy 推理 SDK 用户手册

V0.1

目录

文档记录

修改版本	修改内容	修改人	时间
V0.1	初始化文档	李桂友	2023.07.10

1. 概述

gddeploy 是共达地面向模型部署推理场景自创的推理部署框架，具有简易、多层次接口、推理性能高效、对接多款推理芯片等优点，以满足不同客户需求。

目前已支持硬件平台和算法列表：

硬件\算法	分类	检测	关键点	分割	OCR	动作	多模态	人脸识别	车牌识别
Nvidia									
算能	✔	✔	✔	✔	✔	✔	✔	✔	✔

1.1 gddeploy SDK 包说明

包含目录和说明：

```
bin    // 测试可用的可执行文件
dockerfile      //搭建环境使用的 docker 和安装脚本
docs
lib     // gddeploy 相关库文件
include // gddeploy 相关头文件，包含 app/api/common
sample
script  // 方便使用的脚本文件
thirdparty
    opencv
    ffmpeg
tools
    gtx_maker//生成 SN 码工具
version.txt
README.md
```

注：

1）由于硬件厂商所提供的的 SDK 包过大，因此不放入 thirdparty，用户可使用 1.3 章节使用 dockerfile 构建环境获取硬件厂商所需要的依赖包和编译运行环境；

2）Thirdparty 中的 OpenCV 和 FFmpeg 包版本分别为 4.5.5 和 4.4，用户可复用。如果自己已有别的版本且不可更改，可以将 sample 的 CMakeLists 中的包含 OpenCV 和 FFmpeg 部分声明为 private 进行链接；

1.2 适用硬件平台环境说明

硬件平台：算能 BM1684 SOC 产品，暂不支持 PCIE 形态和 BM1684X 产品

编译链：aarch64-linux-gnu-g++7.5，cmake 3.20
系统版本:

```
VERSION: 2.5.0
KernelVersion：Linux bm1684 4.9.38-bm1684-v10.3.0-00528-g8be6792 #2 SMP
Sun Jan 30 07:12:27 CST 2022 aarch64 GNU/Linux
HWVersion: 0x03
MCUVersion: 0x34
```

注：
1）算能官网提供 SDK 早期版本为 2.3~3.0，后续采用日期作为版本发布，如 V22.12.01、V23.05.01。此处**兼容版本为 2.5.0 及其以后版本**；
2）另外一种常见版本命名方式为 libsophon 版本方式，使用命令 bm_version 可查看，如下：

```
sophon-soc-libsophon : 0.4.4
sophon-soc-libsophon-dev : 0.4.4
sophon-mw-soc-sophon-ffmpeg : 0.5.1
sophon-mw-soc-sophon-opencv : 0.5.1
BL2 v2.7(release):308dcca Built : 19:56:15, Dec 27 2022
BL31 v2.7(release):308dcca Built : 19:56:15, Dec 27 2022
U-Boot 2022.10 308dcca (Dec 27 2022 - 19:56:10 +0800) Sophon BM1684
KernelVersion：Linux bm1684 5.4.217-bm1684-gc1ab88d2690f #1 SMP Tue Dec
27 19:56:19 CST 2022 aarch64 aarch64 aarch64 GNU/Linux
HWVersion: 0x25
MCUVersion: 0x05
```

2. 快速入门

为方便用户能快速试用看到算法效果，可参考本章节快速搭建环境和试用命令行的方式运行算法模型，查看效果

2.1 环境搭建说明

环境分为编译环境和运行环境；
编译环境为本机构建 docker 镜像，安装对应的编译链和对应硬件厂商 SDK 包，

```
docker build -f docker/bmnn_build.Dockerfile -t
devops.io:12580/lgy/test/gddeploy/bmnn/build/3.0:v0.2 .
```

创建容器：

```
docker run -it --rm -v /root/work/gddeploy/install_bmnn/:/root/bmnn
evops.io:12580/lgy/test/gddeploy/bmnn/build/3.0:v0.2
```

或者可直接使用共达地提供的编译环境 docker 镜像

```
docker pull devops.io:12580/lgy/test/gddeploy/bmnn/build/3.0:v0.2
```

2.2 编译程序

准备好编译环境容器后，可以进行源码编译，已提供编译脚本在
script/build.sh
可执行运行：

```
bash script/build.sh
```

2.3 运行程序

2.3.1 运行环境：

设备端运行 gddeploy 可执行程序所需要环境，客户自行选择直接命令行方式运行还是需要构建 docker 环境运行；

拷贝 gddeploy SDK 包到目标机器上并解压，运行脚本

```
# 声明环境变量
source script/env.sh
```

如果存在编译的系统库和运行自带的 C++std 库版本不兼容问题，建议运行容器环境，运行容器环境的命令如下：

```
docker run -it --name test --privileged -v $PWD:/workspace -v
/system/:/system ubuntu:20.04
```

注意算能 3.0 版本之前的版本，算能 SDK 包放在/system/lib 目录，以后的版本在/opt/sophon 目录；运行容器需要映射目录进去；

进入容器后，运行脚本

```
cd /workspace/
source script/env.sh
```

2.3.2 获取 gxt 文件

运行工具获取 gtx 文件

```
./tools/gxt_maker
# 将打印：
# Usage: ./tools/gxt_maker [gxt_file_path]
# SN:HQDZKM6BJAABF0641, UUID :20184e8c-cb15-3014-6b06-558b215168ab
# Gtx file will be save in: .//20184e8c-cb15-3014-6b06-558b215168ab.gxt
```

请将得到的 gxt 文件上传到共达地训练平台，得到 model.gem 和 license 文件，可以将 model.gem 和 license 文件放到 data/models/目录下

2.3.3 运行和测试样例

```
# 查看 pic 用法
./bin/sample_runner_pic -h
# Options:
#   -h [ --help ]         Help screen
#   --model arg           model file path
#   --license arg         model license file path
#   --pic-path arg        pic file full path or just pic path
#   --save-pic arg        save file path

# 运行样例
./bin/sample_runner_pic --model ./data/models/model.gem
--license ./data/models/license
--pic-path ./data/pic/baidu_person/images/
--save-pic ./data/pic/baidu_person/preds/
```

查看 video 用法

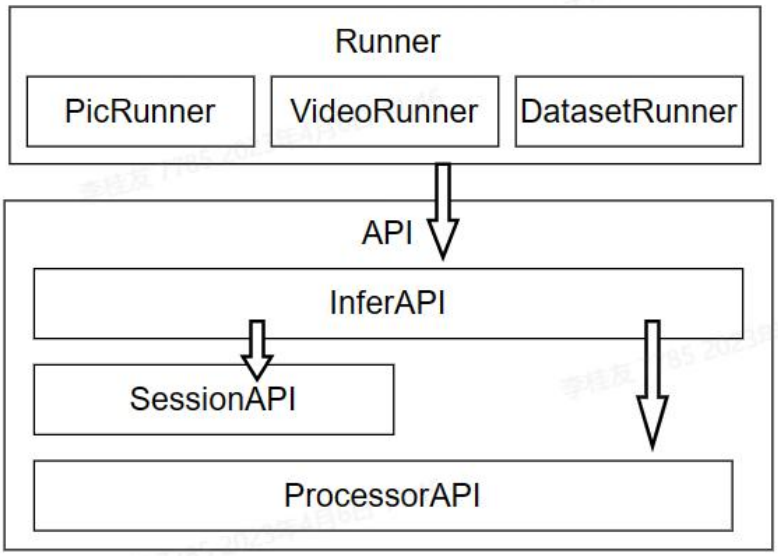
```
./bin/sample_runner_video -h
# Options:
#   -h [ --help ]           Help screen
#   --model arg             model file path
#   --license arg           model license file path
#   --video-path arg        video file path
#   --multi-stream arg (=1) multi stream
#   --is-save arg (=1)      is save result pic
#   --save-pic arg          model file path
# 注意：由于边解码边保存图片会造成资源浪费，--save-pic 功能目前没有开放，用户需要有需要可以修改 sample/sample_runner_video.cpp 文件后重新编
```

```
译

# 运行样例
./bin/sample_runner_video --model ./data/models/model.gem
--license ./data/models/license --video-path
rtsp://admin:gddi1234@10.13.0.104:554/h264/ch1/main/av_stream
--multi-stream 4 --is-save 0 --save-pic .
```

3. API 使用说明

SDK 提供 Runner 和 API 两个层次的接口，软件包含示意图如下：



Runner 为针对某个目的功能的类定义，目前提供图片推理、视频流推理、数据集测试准确率三个功能；

API 主要是单帧数据进行模型推理功能，针对不同客户功能要求进行粗细粒度分层；分别是：

1)针对不关心底层和资源管理的客户，使用 InferSync/InferAsync 等简易接口；

2）针对本身也有其他算法等需要管理硬件资源，使用 SessionAPI 接口，其中包含 InferService/context/session 接口设置和控制资源使用情况；

3）针对有开发能力比较强的客户，提供 ProcessorAPI 接口，包含算法的 pre/infer/post 三个单元，用户自行决定调用的调度策略；

针对模型计算量大小差异，推荐使用 API 如下：

1）算力小，一般是端侧推理，建议使用简约接口和 pre/infer/post 接口

2）算力中等以上：建议使用资源接口

3.1 Runner 说明

Runner 是针对某一特定目的可直接运行程序，比如读取图片和推理，读取视频流和推理，读取数据集和推理获取结果进行准确率测试。API 是指单纯的算法推理。Runner 是在 API 基础上做的一些方便用户快速搭建使用的方式。源码也在 release 的 sample/app 目录，用户可自行修改；

包括如下 runner：
PicRunner：推理单图片接口，有同步和异步接口，包含图片解码->推理->结果解析和画图
VideoRunner：包含 ffmpeg 和 opencv 版本，可进行一路视频流解码和推理或者多路重复推理
DatasetRunner：输入推理数据集图片，获取结果，视模型类型最后保存 json 文件或者直接打印准确率

SDK 包中 bin 目录已有编译好可执行程序,可直接在每个可执行程序--help 查看用法

以图片推理作为例子解释推理基本流程：

```
    cv::Mat in_mat = cv::imread(pic_path);    // 读取图片文件解码

    gddeploy::BufSurfWrapperPtr surf;

#if WITH_BM1684
    bm_image img;
    cv::bmcv::toBMI(in_mat, &img, true);    // bmnn 一般使用 bm_image 结构体进行后续操作
    convertBmImage2BufSurface(img, surf, false);    // bm_image 转为 surface
#else
    convertMat2BufSurface(in_mat, surf, true);
#endif

    // 创建输入输出对象空间
    gddeploy::PackagePtr in = gddeploy::Package::Create(1);
    in->data[0]->Set(surf);

    gddeploy::PackagePtr out = gddeploy::Package::Create(1);

    // 调用 InferAPI 接口进行推理
    infer_api_.InferSync(in, out);

    // 取出结果中的 MetaData，解析结果并打印结果
```

```
    gddeploy::InferResult result =
out->data[0]->GetMetaData<gddeploy::InferResult>();
    PrintResult(result);
```

可以看到调用的基本流程如下：
1）Init，详看 2.2 部分的 Init 所需参数；一般是读取模型和一些全局设置；
2）读取多媒体解码得到一帧数据；
3）转换为 surface 内存格式；
4）创建推理的输入输出 package 空间；
5）推理帧数据；
6）取出结果，解析结果，并进行结果的打印或者绘画结果到原图等操作；

可以看到模型推理输入的结构体为 Package，详细可看 2.4 结构体定义说明；设计考量主要是为方便用户输入一帧或者多帧数据的灵活性和多 batch 一般对于推理性能的提高。而推理结果也会在对应的 data 变量的 MetaData 可以获取；

前端解码后结构体需要进行转为 gddeploy::BufSurfaceWrapper 类型，再赋值 package 中 data
OpenCV 的 Mat 转 Package，参考 type_convert.cpp 文件
FFmpeg 的 AVframe 转 Package，参考 type_convert.cpp 文件
注意：
如果 surface 的 data_ptr 直接指向原数据帧空间，请务必存活整个周期，否则请新建和拷贝；

3.2 API 调用流程：

如上所述，API 功能是完成帧数据的推理功能，不包含前端解码和后续开发；有三种不同层次粗细粒度，均需要用户自行完成图片/视频解码后转为特定内存结构进行输入。

3.2.1 Infer API

对应 sample/infer_api.cpp 文件，本质为调用 Session API 和 Processor API 实现的进一步封装更简易的接口，可通过 Init 函数设置使用哪种底层接口。

```
class InferAPI{
public:
    InferAPI();
    ~InferAPI();

    // api_type: 选择底层的 api 接口为 processor 或者 session api，区别在于有无预分配空间
```

```
    void Init(std::string config, std::string model_path, ENUM_API_TYPE
api_type = ENUM_API_PROCESSOR_API);

    // 同步接口
    int InferSync(const gddeploy::PackagePtr &in, gddeploy::PackagePtr
&out);  //opencv4 可支持解码图片格式

    // 异步接口
    void SetCallback(InferAsyncCallback cb);
    int InferAsync(const gddeploy::PackagePtr &in, InferAsyncCallback
cb = nullptr, int timeout = 0);
    int WaitTaskDone(const std::string& tag="");

    std::string GetModelType();

private:
    std::shared_ptr<InferAPIPrivate> priv_;
};
```

注意：Processor API 为单算法的前处理、推理、后处理，因此不会有异步接口功能，如果希望使用异步接口需要 Init 函数选用 ENUM_API_SESSION_API 参数；

3.2.2 Session API

Session API 一般针对中高算力，可同时推理多路视频流或者高吞吐量场景。需要在 Init 阶段进行资源的提前划分，内存预分配等操作，而且推理阶段尽可能是异构流水线并行计算，以达到最高的使用性能。因此建议使用异步接口，使用过程中需要多次调整参数

```
class SessionAPI{
public:
    SessionAPI();

    int Init(const std::string config, const std::string model_path,
const std::string properties_path = "");
    int Init(const SessionAPI_Param &config, const std::string
model_path, const std::string properties_path = "");

    // 同步接口
    int InferSync(const gddeploy::PackagePtr &in, gddeploy::PackagePtr
&out);  //opencv4 可支持解码图片格式

    // 异步接口
```

```
    void SetCallback(InferAsyncCallback cb);
    int InferAsync(const gddeploy::PackagePtr &in, InferAsyncCallback
cb = nullptr, int timeout = 0);
    int WaitTaskDone(const std::string& tag="");

    std::string GetModelType();
    std::vector<std::string> GetLabels();

private:
    std::shared_ptr<SessionAPIPrivate> priv_;
};
```

其中需要在 Init 就进行资源参数的设置，定义如下：

```
typedef struct {
    std::string name;
    BatchStrategy strategy;   // 可选 static 和 dynamic
    int batch_timeout;      // dynamic 时可用
    int engine_num;         // 底层可并行运行 engine 个数
    int priority;           // 优先级
    bool show_perf;         // default false
} SessionAPI_Param;
```

对于中高算力硬件设备，大多具有 batch 可以明显提高推理性能的特点，比如 bm1684 的 Batch4 可以达到 Batch1 一样的推理时间，Nvidia 的 TensorRT 中 Batch2~Batch4 有 30%~50%的推理性能提高；因此对于高吞吐量场景尽量选用 dynamic。

3.2.3 Processor API

模型推理基本包含算法前处理、推理、算法后处理。每一部分均为一个 Processor 单元，串起来构建得到算法的 pipeline，相比于 SessionAPI 的优势的比较轻量级，简单，适用于算力较小或推理实时性强的使用场景。

```
class ProcessorAPI{
public:
    ProcessorAPI();
    void Init(std::string config, std::string model_path);

    // 根据模型获取 processor，用于最基础层的接口
    std::vector<ProcessorPtr> GetProcessor();

    std::string GetModelType();

private:
```

```
    std::shared_ptr<ProcessorAPIPriv> priv_;
};
```

可以看到类定义非常的简介，也即是解析模型得到对应算法的前处理、推理、后处理单元，获取到 Processor 对象后，逐个调用输入 package 对象即可推理，可以参考 InferAPI 源码部分如下：

```
int InferAPIPrivate::InferSync(const gddeploy::PackagePtr &in,
gddeploy::PackagePtr &out)
{
    if (api_type_ == ENUM_API_PROCESSOR_API){
        // 4. 循环执行每个 processor 的 Process 函数
        for (auto processor : processors_){
            processor->Process(in);
        }
        out = in;
    }
    return 0;
}
```

3.3 API 详细说明

详看对应头文件说明

3.4 数据结构定义：

gddeploy::PackagePtr 说明：
异步接口时，如果直接赋值 data_ptr 指向空间，请务必存活整个周期，也可让 gddeploy::BufSurfaceWrapper 托管释放空间

```
  struct Package
  {
    /// a batch of data, origin data
    BatchData data;

    /// private member, intermediate storage，可能会作为前处理和推理后数据临时存储
    InferDataPtr predict_io{nullptr};

    /// tag of this package (such as stream_id, client ip, etc.)
    std::string tag;
```

```
    /// perf statistics of one request
    std::map<std::string, float> perf;

    /// private member
    int64_t priority;

    static std::shared_ptr<Package> Create(uint32_t data_num, const
std::string &tag = "") noexcept
    {
      auto ret = std::make_shared<Package>();
      ret->data.reserve(data_num);
      for (uint32_t idx = 0; idx < data_num; ++idx)
      {
        ret->data.emplace_back(new InferData);
      }
      ret->tag = tag;
      return ret;
    }
  };
```

使用技巧：

data 保存了输入的帧数据，如果需要多模型串联，可以重复从中裁剪帧数据继续送入第二模型处理

3.5 结果解析说明：

具体详细说明请看 result_def.h

```
typedef struct {
    std::vector<int> result_type;
    DetectResult detect_result;
    DetectPoseResult detect_pose_result;
    ClassifyResult classify_result;
    SegResult seg_result;
    ImageRetrievalResult image_retrieval_result;
    FaceRetrievalResult face_retrieval_result;
    OcrDetectResult ocr_detect_result;
    OcrRecResult ocr_rec_result;
    void *user_data;
}InferResult;
```

在实际运行过程中需要多个模型串联，为使后需要模型可以用上一模型的结果，设计为把各类算法结果统一起来，因此解析的时候首先读取 result_type，然后解析对应算法结构体；

3.6 内存管理说明

内存/显存采用 surface 结构体管理的方式，定义解析如下：

```
/**
 * Holds information about a single buffer in a batch.
 */
typedef struct BufSurfaceParams {
  /** Holds the width of the buffer. */
  uint32_t width;
  /** Holds the height of the buffer. */
  uint32_t height;
  /** Holds the pitch of the buffer. */
  uint32_t pitch;
  /** Holds the color format of the buffer. */
  BufSurfaceColorFormat color_format;

  /** Holds the amount of allocated memory. */
  uint32_t data_size;

  /** Holds a pointer to allocated memory. */
  void * data_ptr;

  /** Holds a pointer to a CPU mapped buffer.
  Valid only for CNEDK_BUF_MEM_UNIFIED* and CNEDK_BUF_MEM_VB* */
  void * mapped_data_ptr;

  /** Holds planewise information (width, height, pitch, offset, etc.).
*/
  BufSurfacePlaneParams plane_params;

  void * _reserved[CNEDK_PADDING_LENGTH];
} BufSurfaceParams;

/**
 * Holds information about batched buffers.
 */
typedef struct BufSurface {
  /** Holds type of memory for buffers in the batch. */
  BufSurfaceMemType mem_type;

  /** Holds a Device ID. */
  uint32_t device_id;
```

```
  /** Holds the batch size. */
  uint32_t batch_size;
  /** Holds the number valid and filled buffers. Initialized to zero when
   an instance of the structure is created. */
  uint32_t num_filled;

  /** Holds an "is contiguous" flag. If set, memory allocated for the batch
  is contiguous. Not valid for CNEDK_BUF_MEM_VB on CE3226 */
  bool is_contiguous;

  /** Holds a pointer to an array of batched buffers. */
  BufSurfaceParams *surface_list;

  /** Holds a pointer to the buffer pool context */
  void *opaque;

  /** Holds the timestamp for video image, valid only for batch_size ==
1 */
  uint64_t pts;

  void * _reserved[CNEDK_PADDING_LENGTH];
} BufSurface;
```

补充说明：
这里的 batch_size 决定 surface_list 有多少个，而 BufSurfaceParams 中 data_ptr 都指向对于 batch 的地址,data_size 一般为 CHW*sizeof(pixel_size) 大小，BufSurfacePlaneParams 中的 Plane 是是指一个通道的数据，也就是 HW 以 batch4 为例，如果数据排布如下：
暂时无法在飞书文档外展示此内容
每个 batch idx 的大小为 channel*height*width,
BufSurfacePlaneParamsoffset 是地址偏移宽度,一般 640*640*sizeof(float),。可以参考 GetColorFormatInfo 函数赋值

内存/显存操作有三个头文件，分别的功能如下：
buf_surface_utils.h：主要是 BufSurfaceWrapper 和 BufPool 类定义，分别是 surface 智能指针管理和内存池作用
buf_surface.h：BufSurfaceService 类及其接口定义，主要是 surface 结构体池和分配
buf_surface_impl.h：接口类，主要是 MemPool 内存池和 MemAllocator 内存分配器定义，MemAllocator 是接口类，各个设备的显存接口需要继承和实现

用户侧分配内存/显存做法：
固定已知内存/显存大小，需要预分配，建议采用 BufPool 方式预分配内存，按需请求获取使用

未知内存/显存大小，临时创建和申请，建议采用 CreateSurface 创建 surface 和 MemAllocator 分配内/显存

4.sample 说明

（后续开放）
多线程流程：
多模型独立：
多模型依赖：
多模型多线程：

5. FAQ

1）有哪些硬件加速技巧，达到最大吞吐量
答：以下操作均有提高性能 tricks，请逐个尝试：

- 解码后映射送入推理，不拷贝
- 选用 SessionAPI 接口，推理时选用异步接口
- Session 参数中增大 batch，一般建议 2~4 即可，同时 timeout 设置 100ms
- 回调函数非阻塞，尽量阻塞的操作通过消息队列等方式在另外线程进行；
- 使用硬件解码编码
- engine 数量，和推理单元数量一致，过低和过高均影响速度
- 同一模型的不同 session，设置参数最好一致，以便最高效使用硬件。

2）如何设置 log 等级：
export SPDLOG_LEVEL=info
目前支持等级：trace/debug/info/warn/err/critical/off